UNIVERSITY OF CAMBRIDGE

xDSL

An Overview

– and why to adopt it –

Tobias Grosser & Chris Vasiladiotis

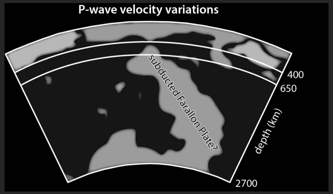
PIs: Nick Brown, Amrey Krause, Michel Steuwer, Gerard Gorman, Paul Kelly

Team: Mathieu Fehr, Sasha Lopoukhine, George Bisbas, Emilien Bauer, Anton Lydike, Nicolai Stawinoga, Alex Rice, Michel Weber, Dalia Shaban, Joren Dumoulin,Théo Degioanni, Christian Ulmann, Prathamesh Tagore, …

ExCALIBUR

Domain-Specific Languages Gain Adoption

xDSL



PSyclone

Stencil X
↓
Targets X

Devito

Stencil Y
↓
Targets Y

✅ Performance ✅ Productivity ✅ Portability

Technical Challenges

- ❌ **Composability**
- ❌ **Code Reuse**
- ❌ **Interoperability**
- ❌ **Longevity**

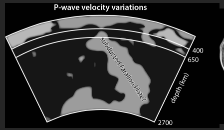
Societal Challenges

- ❌ ***Disjoint Communities***
- ❌ ***Lack of Knowledge Transfer***

ExCALIBUR IO

A Shared Ecosystem for Exascale DSLs

xDSL



...

Application domains, numerical methods

C++ | Fortran | PSyclone | Devito | Other DSLs | ... | New DSLs

Python Toolbox for building MLIR-based DSLs

Multi-level Intermediate Representations (MLIR) permit Composable Abstractions

OpenMP
Loop
GPU

Algorithms: Linalg, Tensor, Stencil, Outer Loop

Python: Sympy, Numpy

HPC: MPI

LLVM-IR

Targets | Targets | Targets | Targets | Targets

Debuggers
Profilers

- ✅ Performance
- ✅ Productivity
- ✅ Portability

- ✅ Composability
- ✅ Interoperability
- ✅ Code Reuse
- ✅ Longevity

- ✅ Connected Communities
- ✅ Knowledge Transfer

Collaborative Support by Academics and Industry via LLVM Community

ExCALIBUR IO

2010

GCC

Microsoft Visual C++

ARM C/C++

Intel ICC

LLVM clang

PathScale C/C++

IBM XL C/C++

HP C++

PGI C++

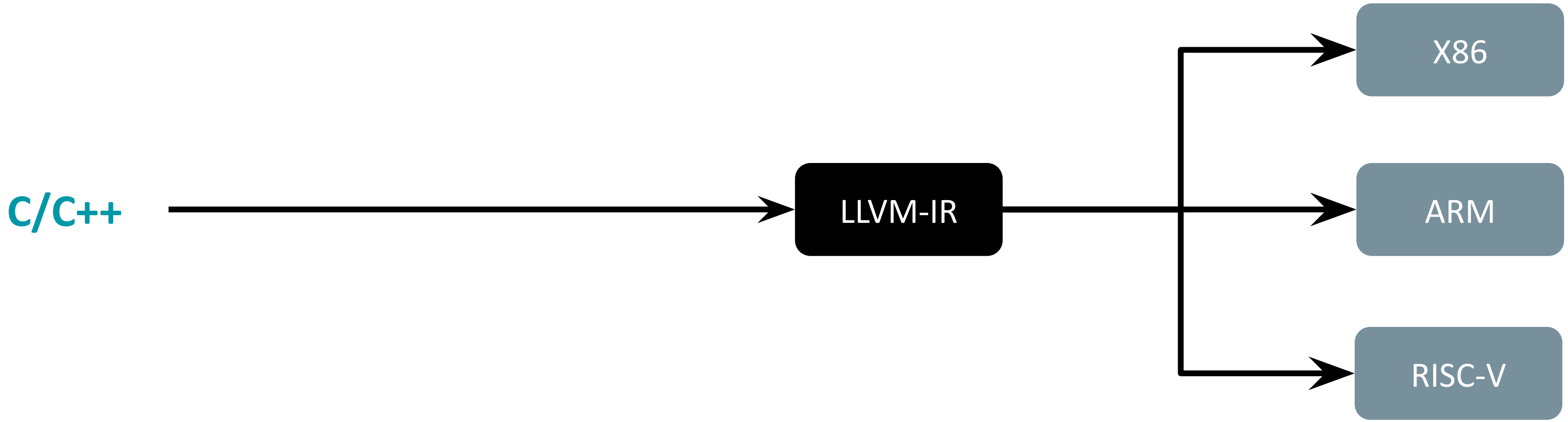
2023

Microsoft Visual C++

GCC

LLVM clang

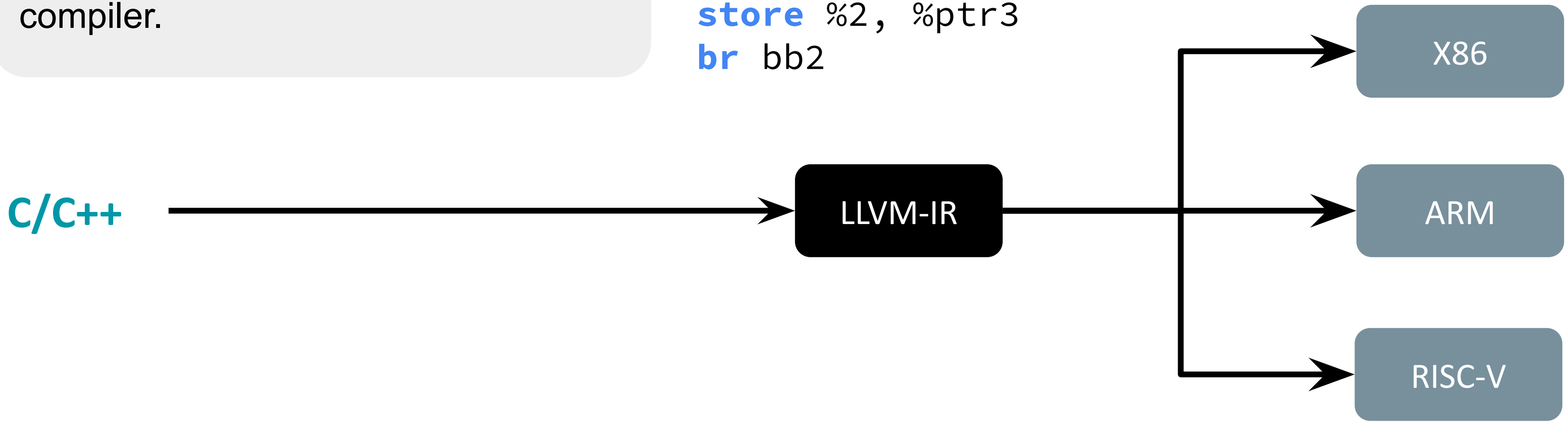
Compiler Pipelines



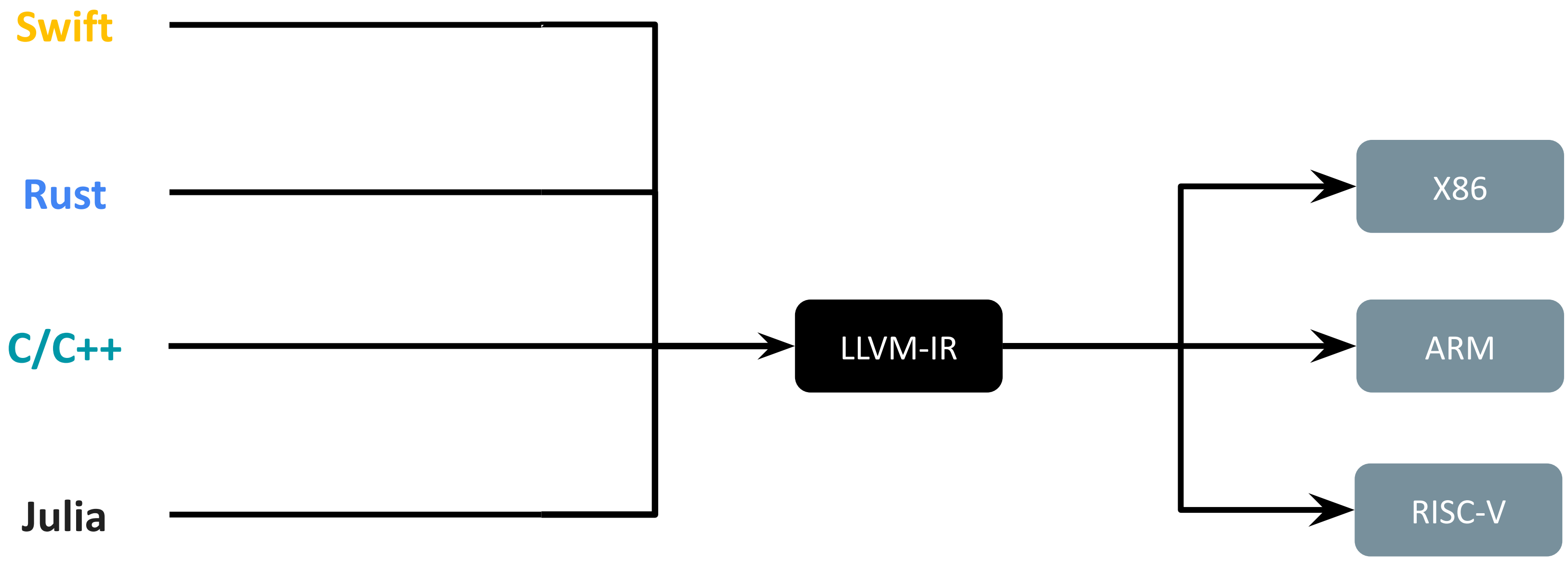
Compiler Pipelines

Intermediate Representations (IRs) are the central abstractions in a compiler.

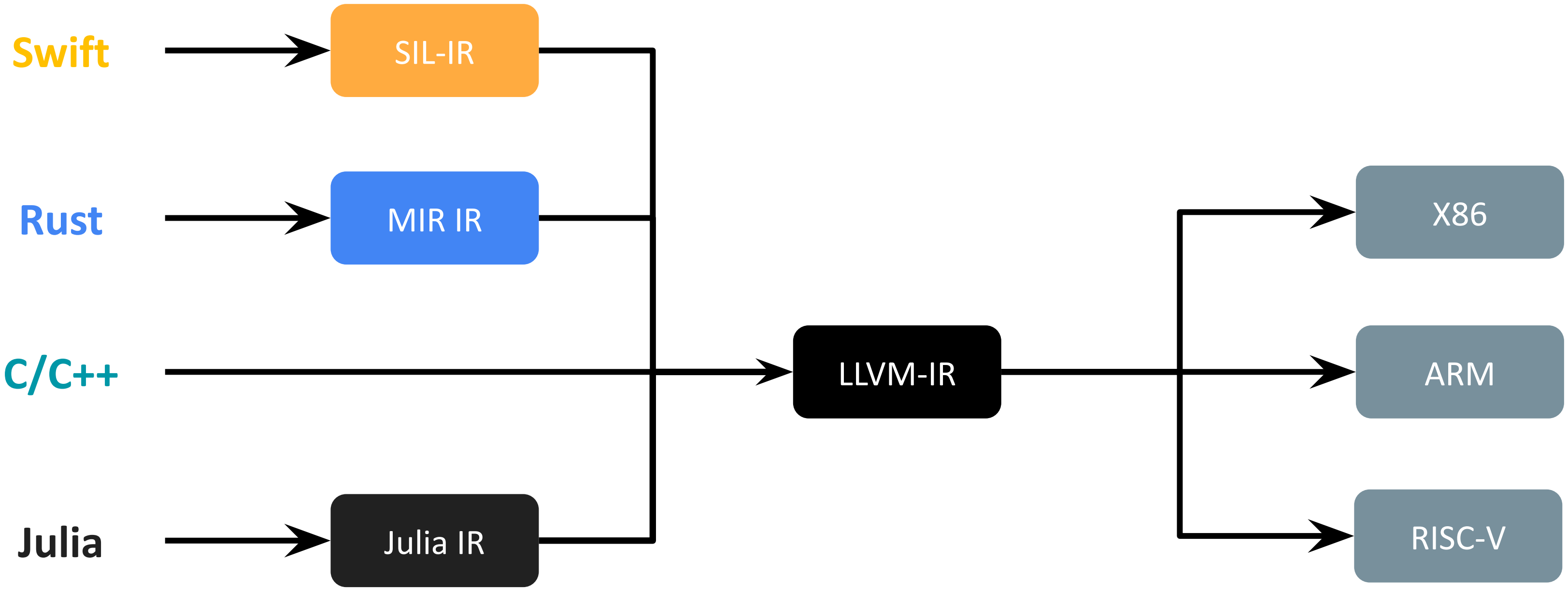
```
%0 = load %ptr
%1 = load %ptr2
%2 = add %0, %1
store %2, %ptr3
br bb2
```



Compiler Pipelines

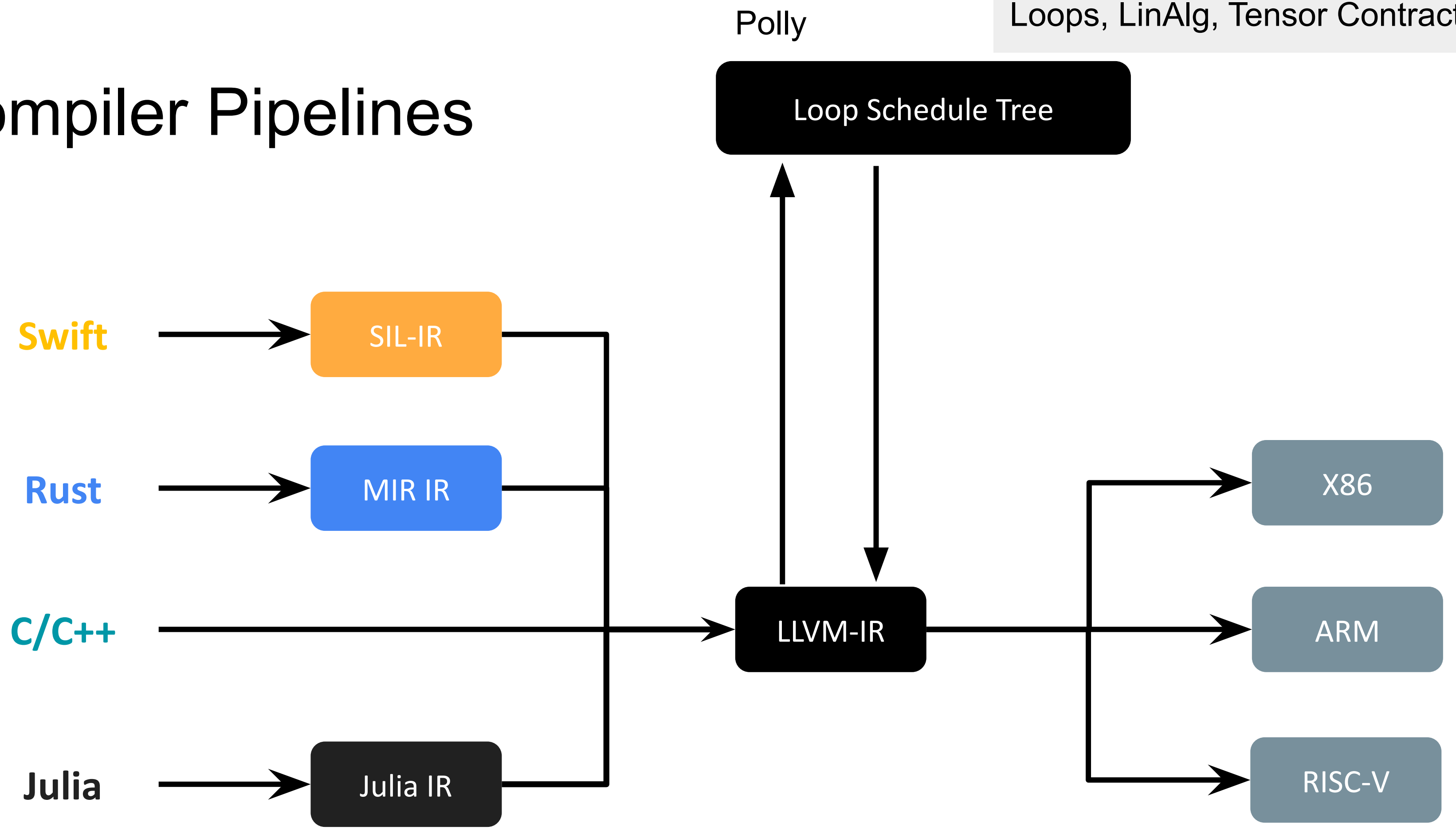


Compiler Pipelines

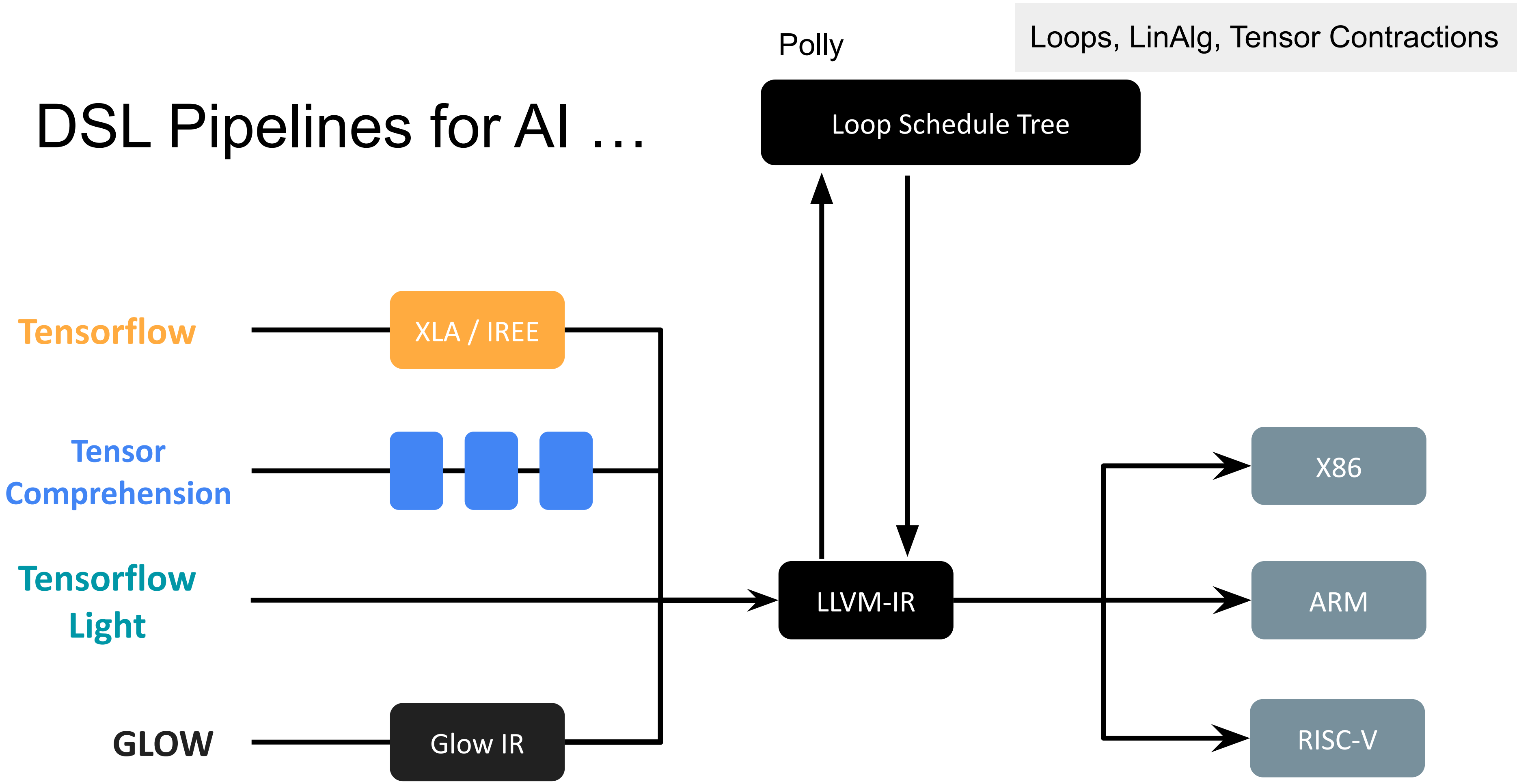


Loops, LinAlg, Tensor Contractions

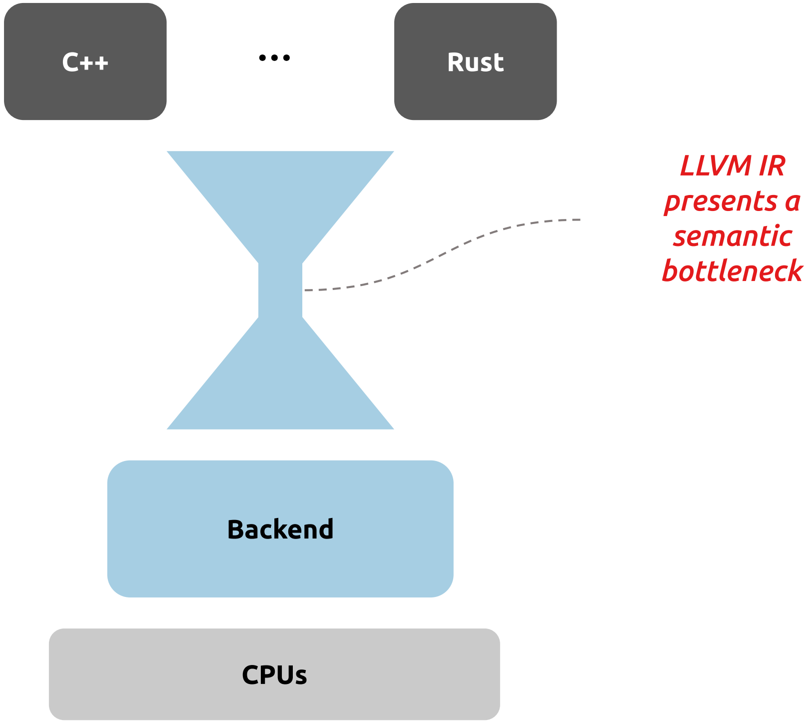
Compiler Pipelines



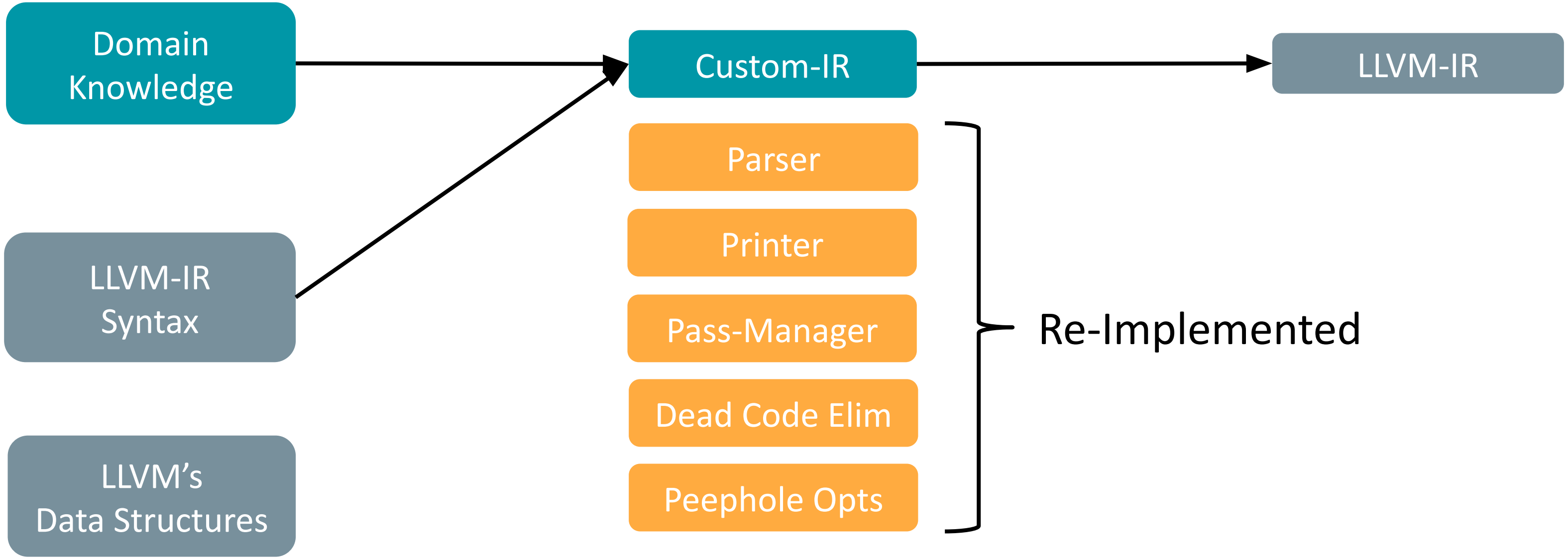
DSL Pipelines for AI …



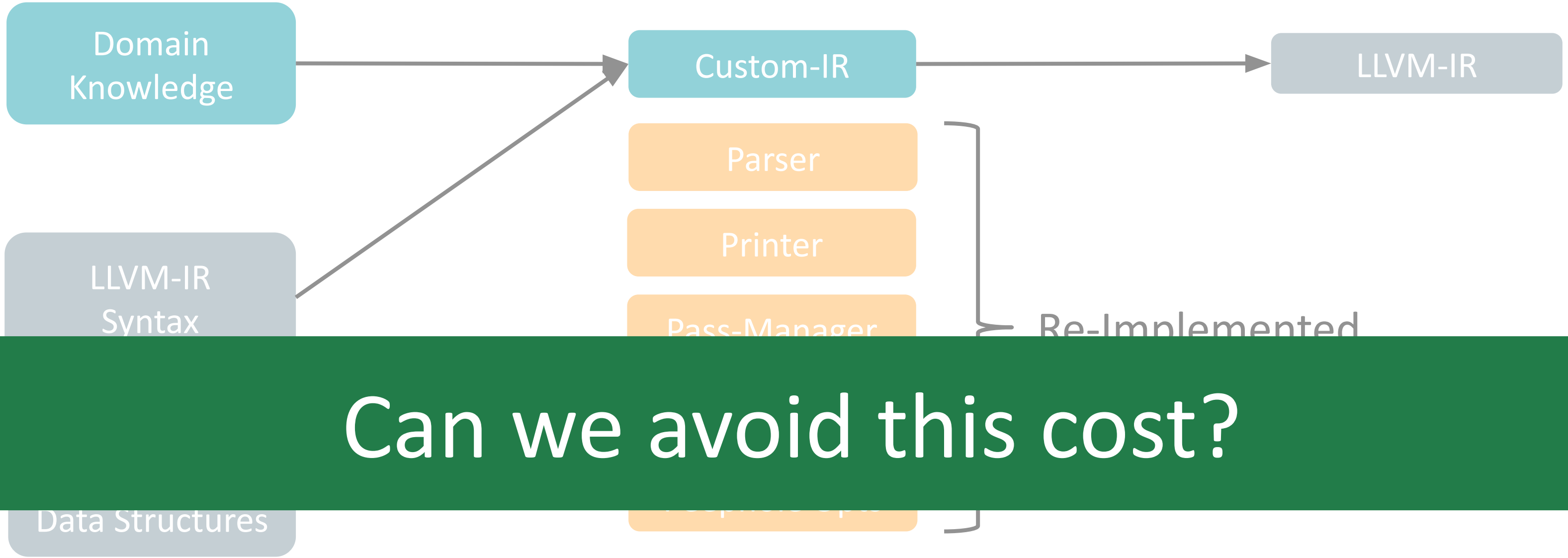
Semantic Bottleneck



How to build a Modern High-Level Compiler IR

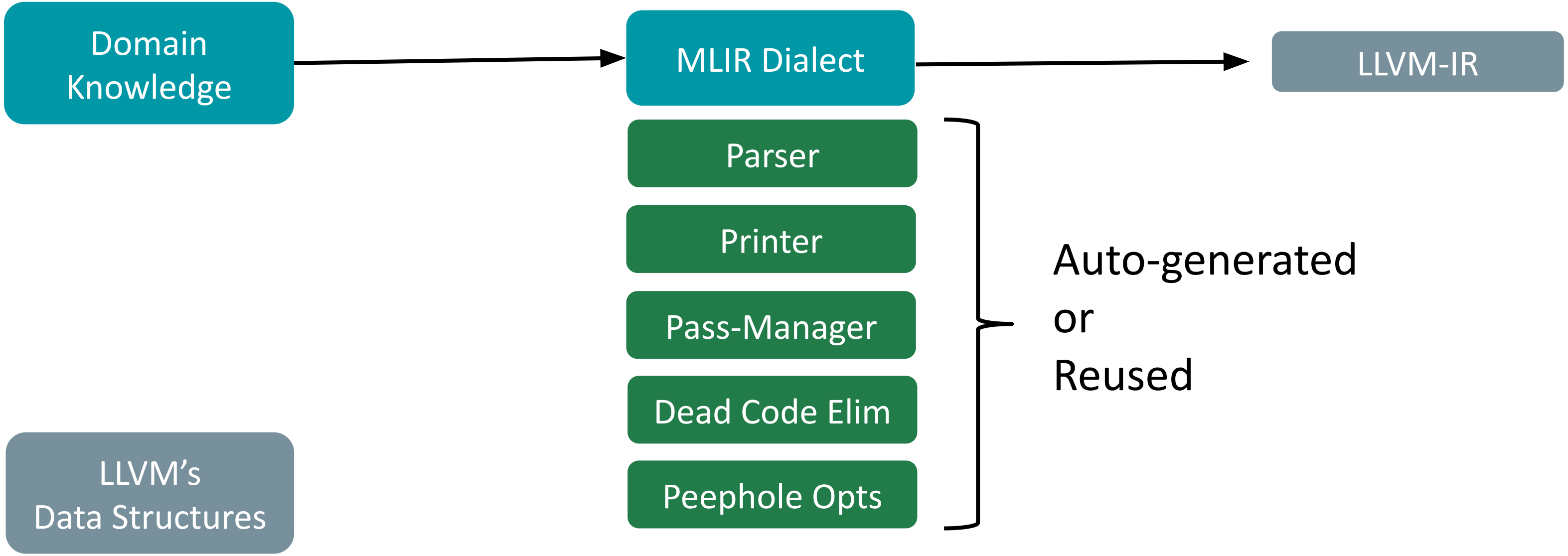


How to build a Modern High-Level Compiler IR

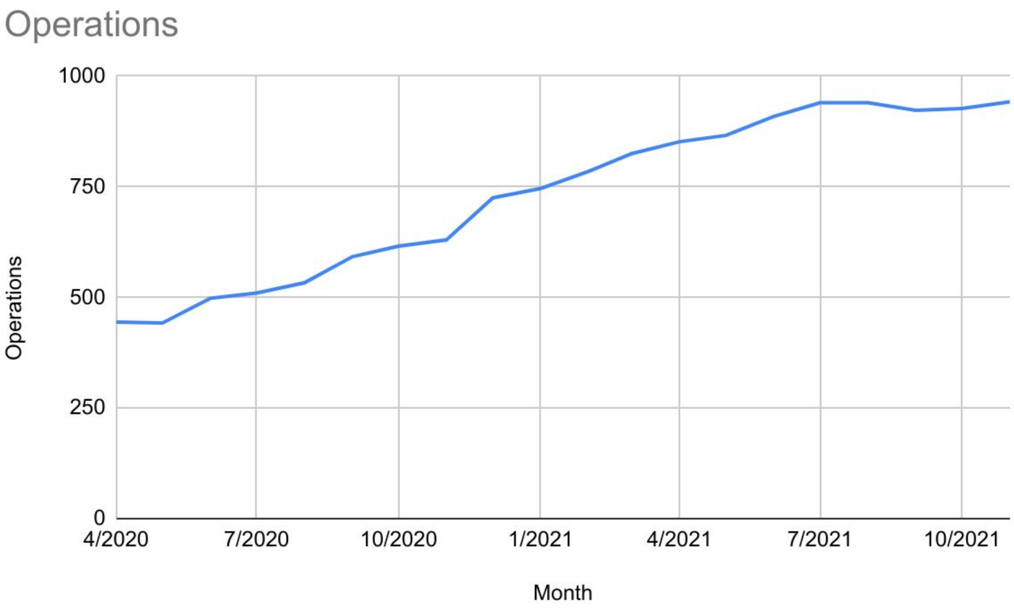


Can we avoid this cost?

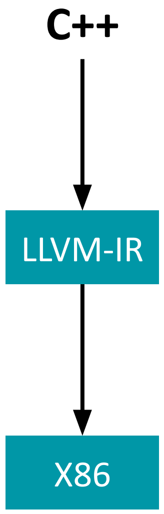
Building a Modern Compiler IR – using MLIR



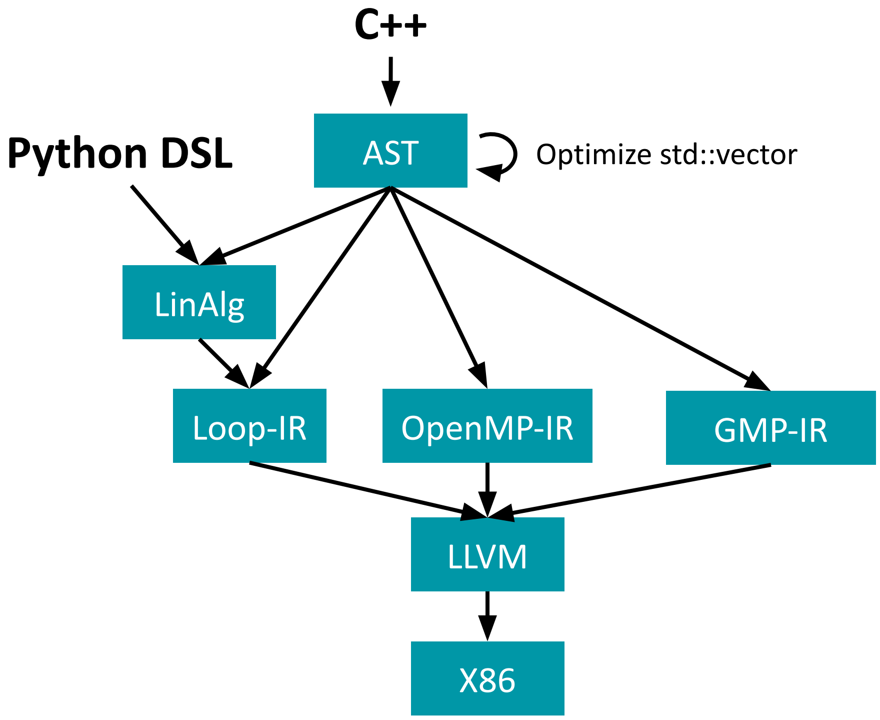
MLIR: A Growing Number of IR Abstractions



Traditional Compilation



New: Multi-Level Rewriting



Defining an IR: Easy or Painful?

C++ Template Meta Programming

TableGen

Build Systems

LLVM Terminology

Compiler Experts:

Defining IRs in MLIR is easy!

vs. gcc, LLVM backends, …

Everybody Else:

Defining IRs in MLIR is confusing!

vs. Python, Domain-Specific Languages, …

Experiment
An SQL Dialect in MLIR

r internship
ate student
QL IR

Teaching 100s of students in 12 weeks compiler design with MLIR?

Impossible!

We spent most of our summer:

- **Teaching TableGen**
- **Fixing Windows Bugs**
- **… but no time for SQL dialect**

Core Concepts

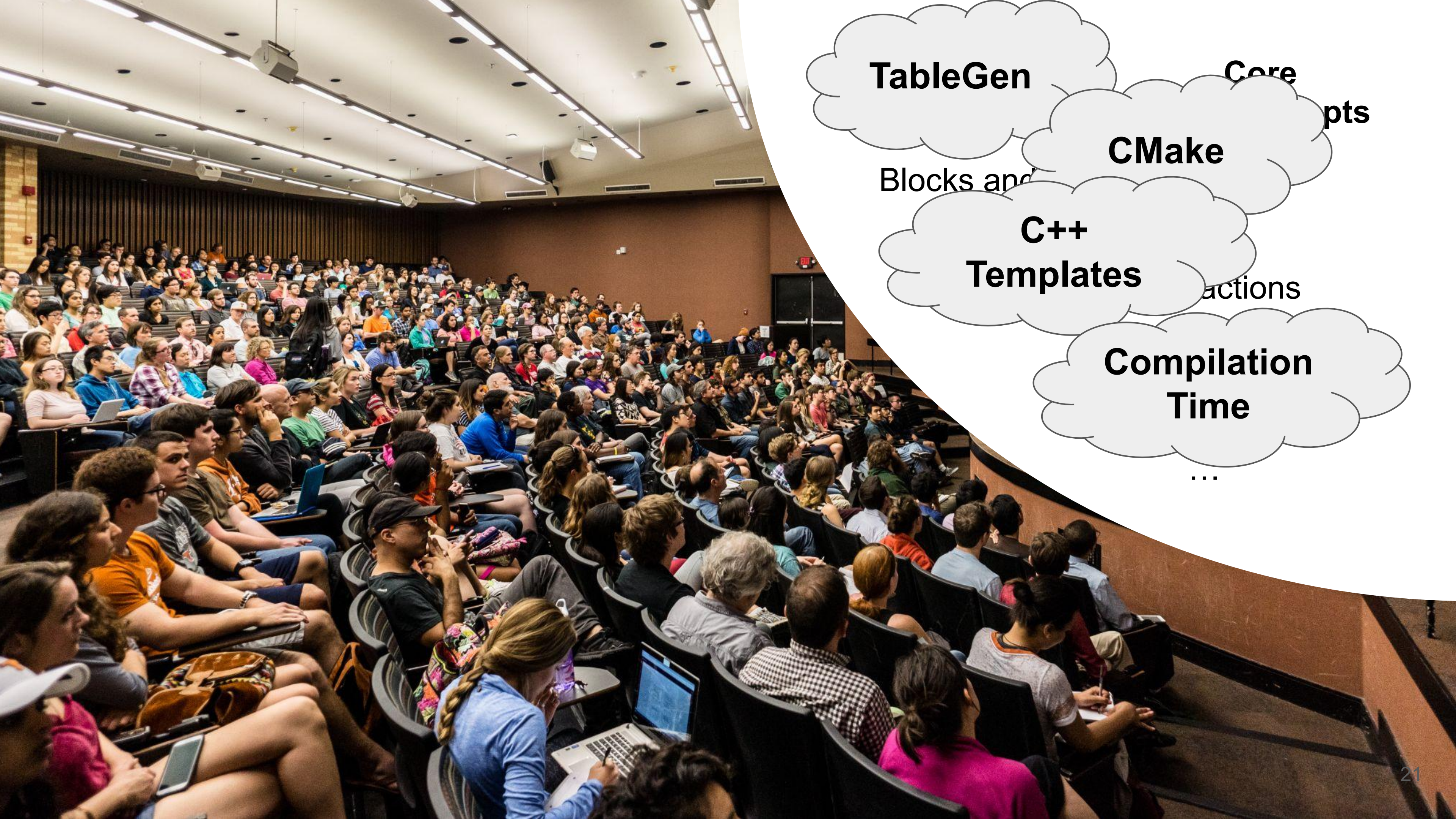
SSA and Operations

Blocks and Regions

Dialects as abstractions

Peephole rewrites

…



TableGen

Core

pts

CMake

Blocks and

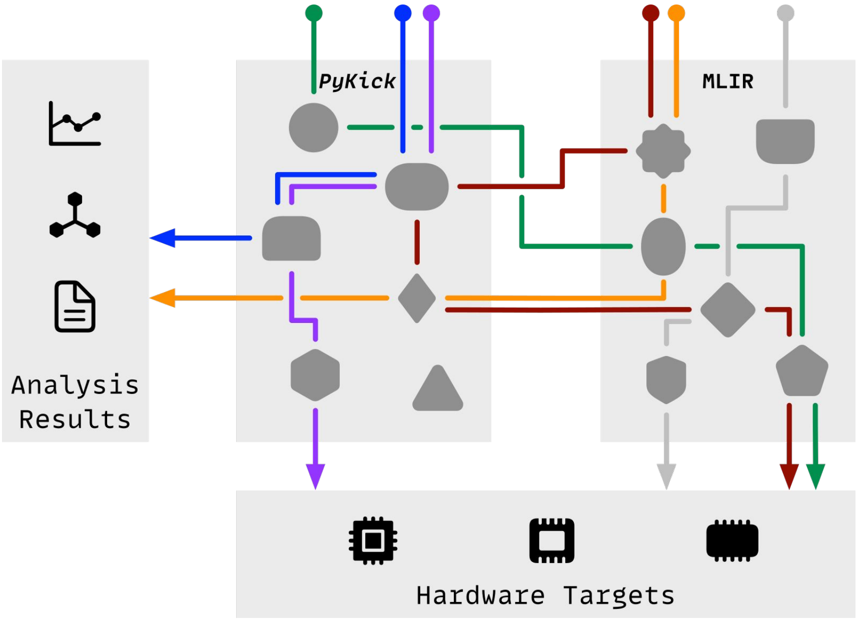
C++
Templates

actions

Compilation
Time

…

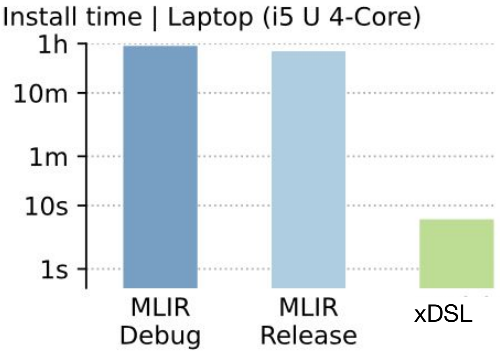
xDSL: a *Sidekick* to MLIR

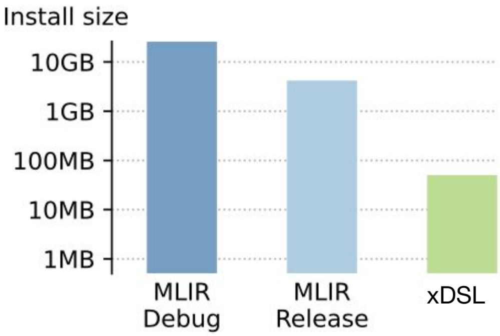


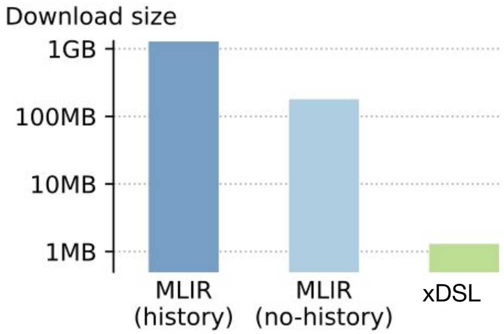
- MLIR core concepts in Python
- Translation of programs from/to MLIR
- Translation of dialects from/to MLIR

xDSL is quick to install (and use?)

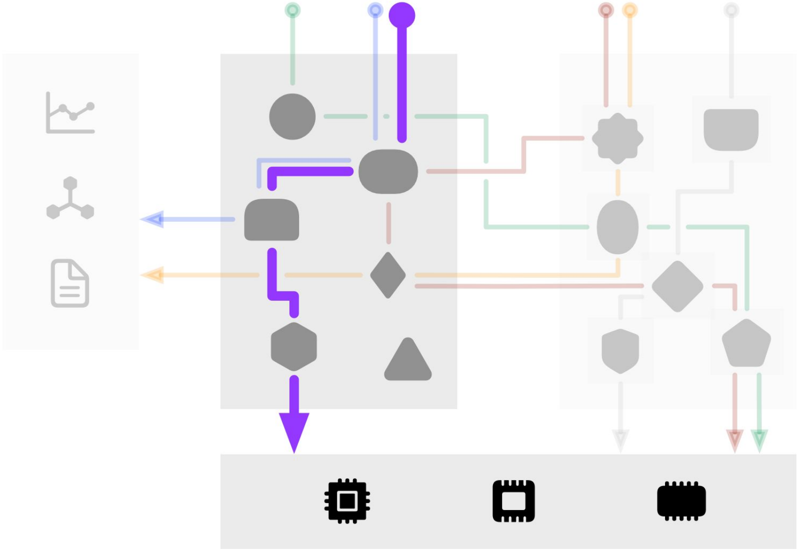
```
pip install xdsl
```



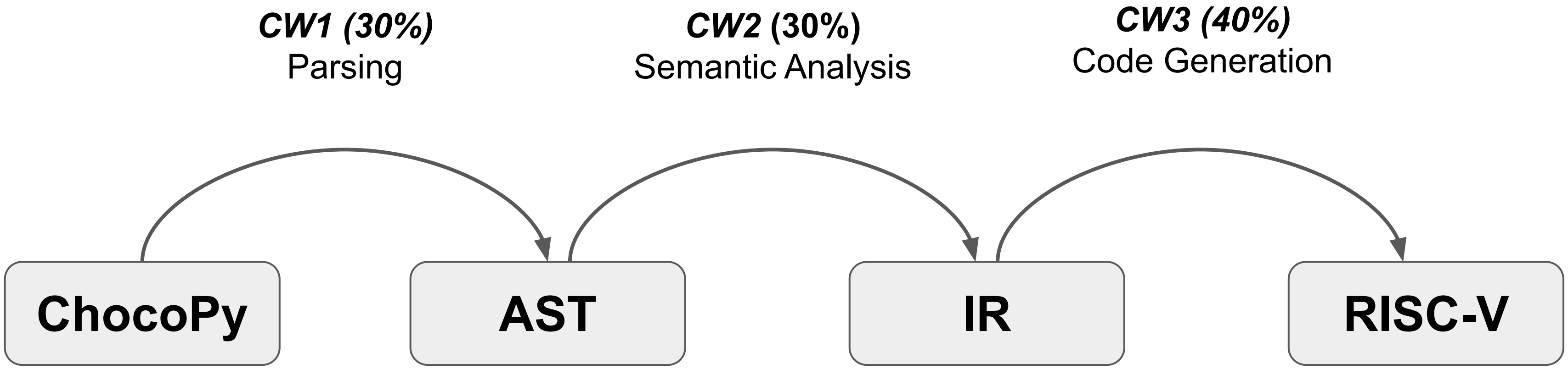




Teaching Compilers in Python

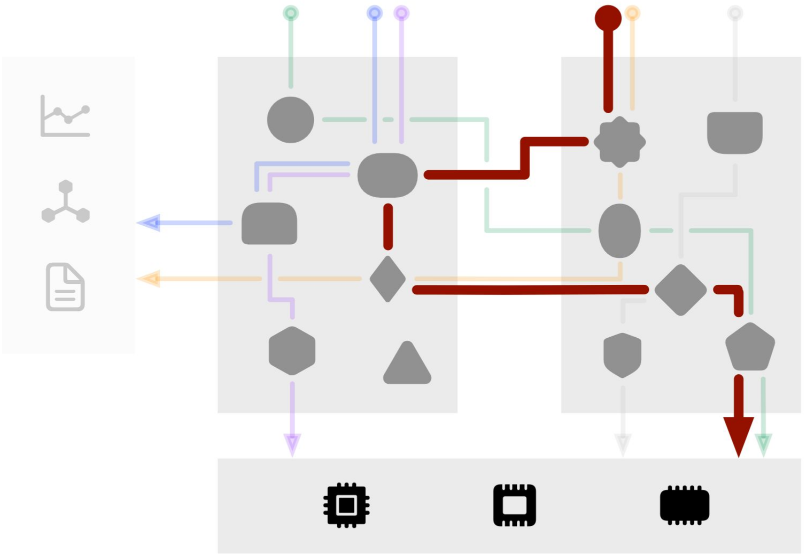


Coursework: A Python to RISC-V Compiler

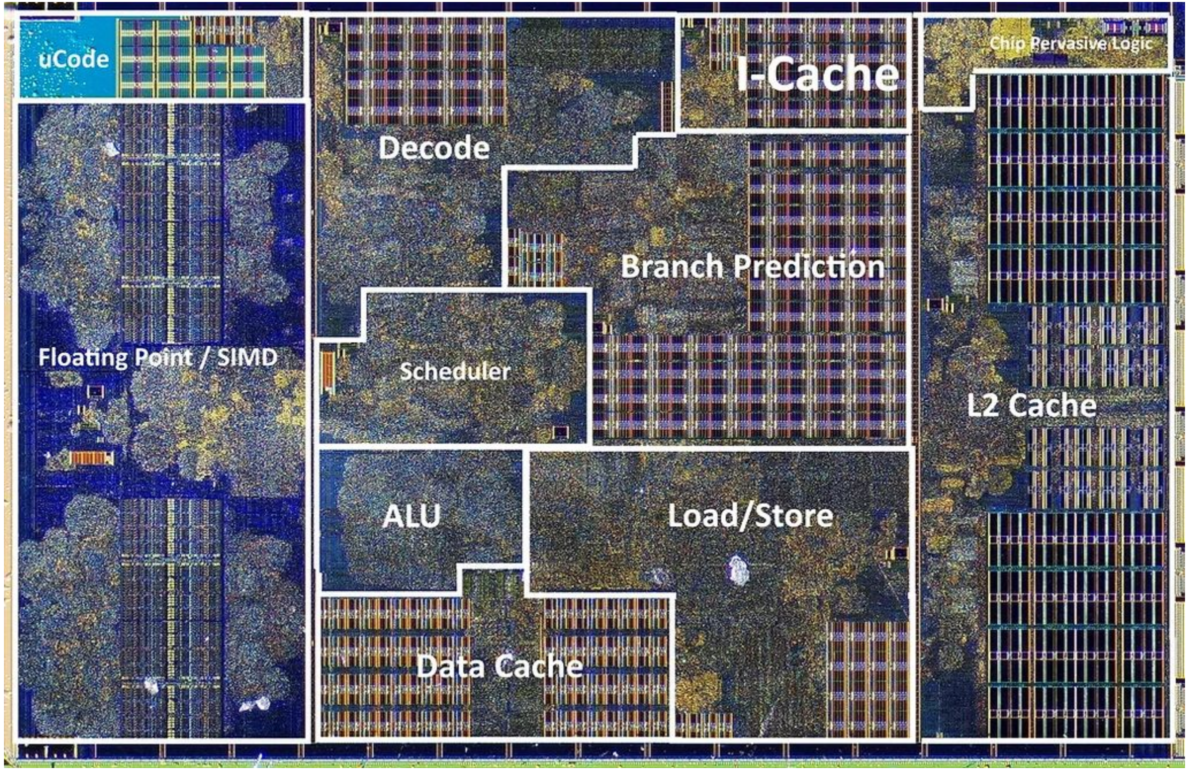


python™ MLIR RISC-V®

Prototyping New Compiler Features

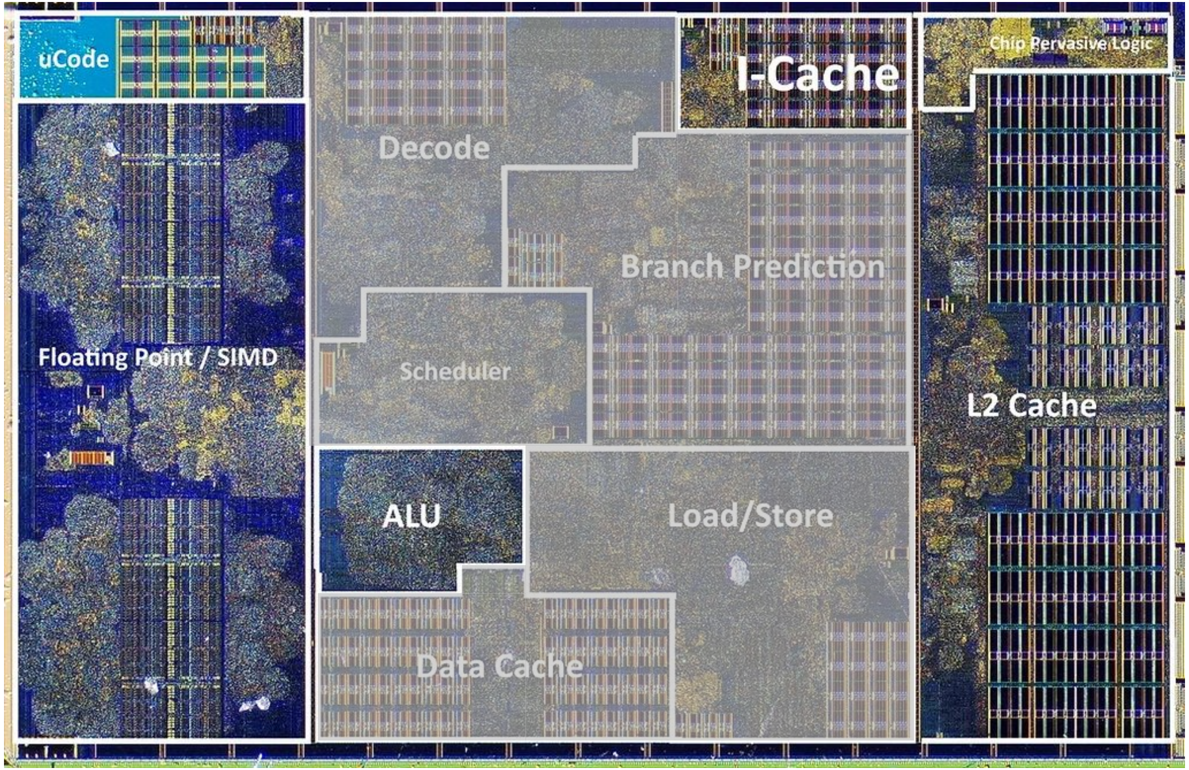


Modern CPUs offer a complex μ-architecture



Die shot by Fritzchens Fritz, Annotation from AMD Zen 2 slides at ISSCC

Linear Algebra does **not** require complex μ-architecture



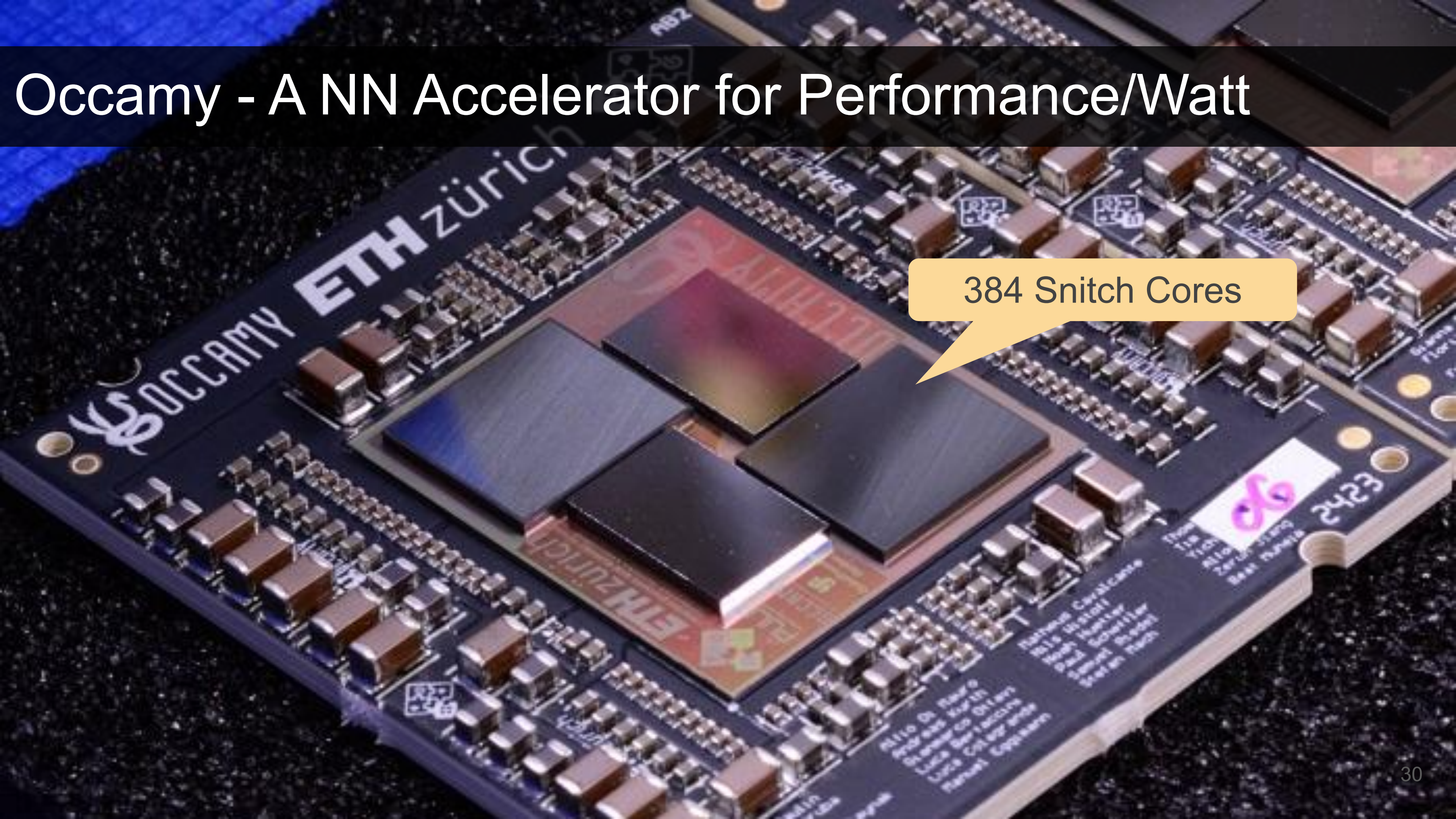
Die shot by Fritzchens Fritz, Annotation from AMD Zen 2 slides at ISSCC

Compute is a Power-Bound Problem

Three Mile Island nuclear reactor to restart to power Microsoft AI operations

Pennsylvania plant was site of most serious nuclear meltdown and radiation leak in US history in 1979

Occamy - A NN Accelerator for Performance/Watt



Snitch Cores Shift Complexity from Hardware to Software

~~Out Of Order Execution~~	Streaming Registers
~~Branch Prediction~~	Hardware Loops
~~Hardware-managed Cache~~	Scratchpad Memory
~~Scalar Instructions~~	Packed SIMD

Perfect Fit for Linear Algebra Workloads

How to utilize the Snitch ISA extensions

ISA Extensions: Standard
0.28 FLOP/instruction

double precision
inner product calculation
in RISC-V assembly

```
.loop:
        fld     fa5, 0(a1)
        fld     fa4, 0(a2)
        fmadd.d fa0, fa5, fa4, fa0
        addi    a2, a2, 8
        addi    a1, a1, 8
        addi    a0, a0, -1
        bnez    a0, .loop
```

How to utilize the Snitch ISA extensions

elided **Standard** ISA instructions

ISA Extensions: Standard
0.28 FLOP/instruction

adding SSRs

ISA Extensions: +SSR
0.66 FLOP/instruction

double precision inner product calculation in RISC-V assembly

```
.loop:
    fld      fa5, 0(a1)
    fld      fa4, 0(a2)
    fmadd.d fa0, fa5, fa4, fa0
    addi     a2, a2, 8
    addi     a1, a1, 8
    addi     a0, a0, -1
    bnez     a0, .loop
```

*added **SSR** configuration*

*enable **SSRs***

*elided explicit memory loads with **SSRs***

*elided pointer arithmetic with **SSRs***

*disable **SSRs***

```
scfgw
scfgw
csrsi    ssrcfg, 1
.loop:
fmadd.d fa0, ft0, ft1, fa0
addi     a0, a0, -1
bnez     a0, .loop
csrsi    ssrcfg, 0
```

How to utilize the Snitch ISA extensions

elided **Standard** ISA instructions

ISA Extensions: Standard
0.28 FLOP/instruction

adding SSRs

ISA Extensions: +SSR
0.66 FLOP/instruction

adding FREP

ISA Extensions: +SSR +FREP
2 FLOP/instruction

double precision inner product calculation in RISC-V assembly

```
.loop:
    fld      fa5, 0(a1)
    fld      fa4, 0(a2)
    fmadd.d fa0, fa5, fa4, fa0
    addi     a2, a2, 8
    addi     a1, a1, 8
    addi     a0, a0, -1
    bnez     a0, .loop
```

*added **SSR** configuration*

*enable **SSRs***

*elided explicit memory loads with **SSRs***

*elided pointer arithmetic with **SSRs***

*disable **SSRs***

```
scfgw
scfgw
csrsi    ssrcfg, 1
.loop:
fmadd.d fa0, ft0, ft1, fa0
addi     a0, a0, -1
bnez     a0, .loop
csrsi    ssrcfg, 0
```

*loop replaced by **frep** instruction*

```
scfgw
scfgw
csrsi    ssrcfg, 1
frep.o   a0, 1, 0, 0
fmadd.d fa0, ft0, ft1, fa0
csrsi    ssrcfg, 0
```

multiply and accumulate
floating-point instruction preserved

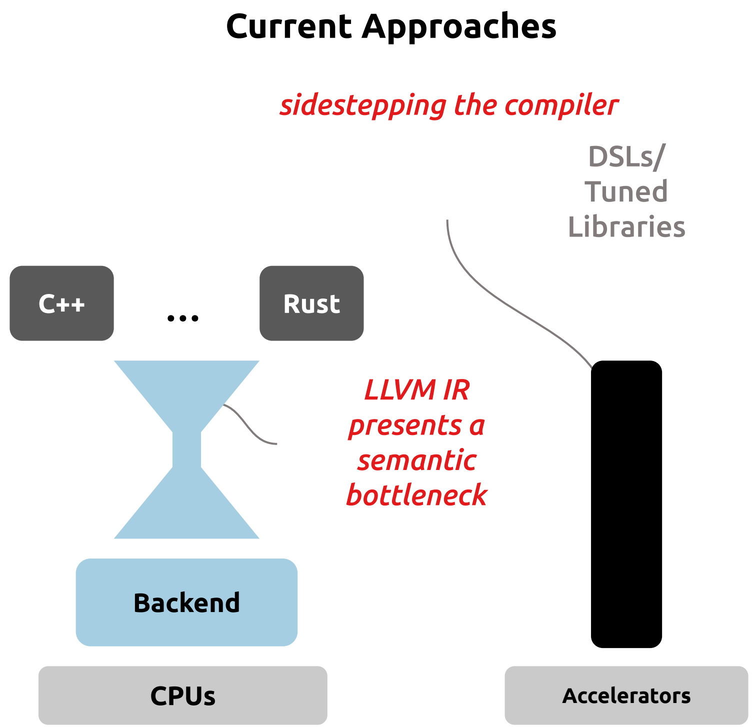
How to **really** utilize the Snitch ISA extensions

```c
const uint32_t m = 8, n = 16;

snrt_ssr_loop_2d(SNRT_SSR_DM_ALL, m, n, sizeof(double) * n, sizeof(double));

snrt_ssr_read(SNRT_SSR_DM0, SNRT_SSR_2D, x);  // ft0
snrt_ssr_read(SNRT_SSR_DM1, SNRT_SSR_2D, y);  // ft1
snrt_ssr_write(SNRT_SSR_DM2, SNRT_SSR_2D, z); // ft2

snrt_ssr_enable();

for (int i = 0; i < m; ++i) {
  for (int j = 0; j < n; ++j) {
    asm volatile("fadd.d ft2, ft0, ft1\n" : : : "ft0", "ft1", "ft2", "memory");
  }
}

snrt_ssr_disable();
```

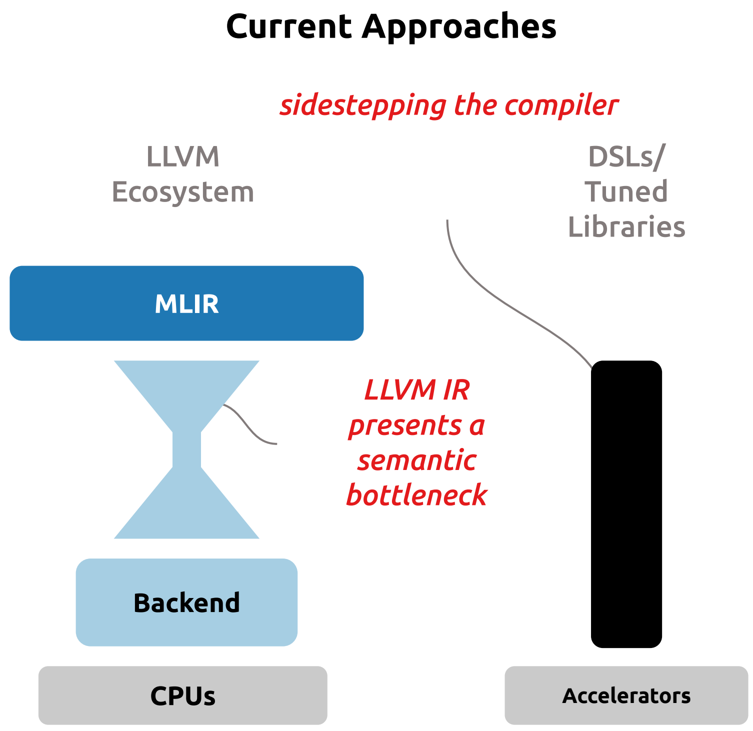
1 extensive use of intrinsics

2 inline assembly

How do we target such accelerators?

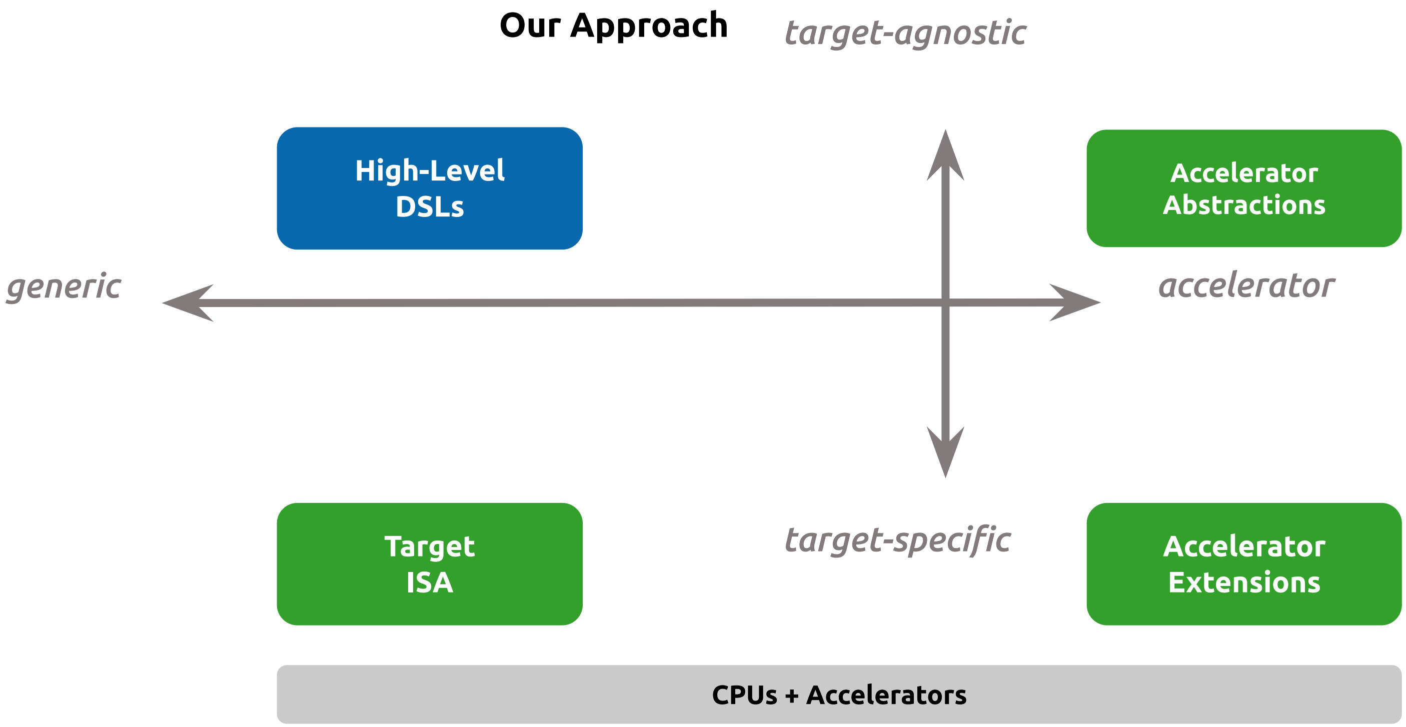


How do we target such accelerators?



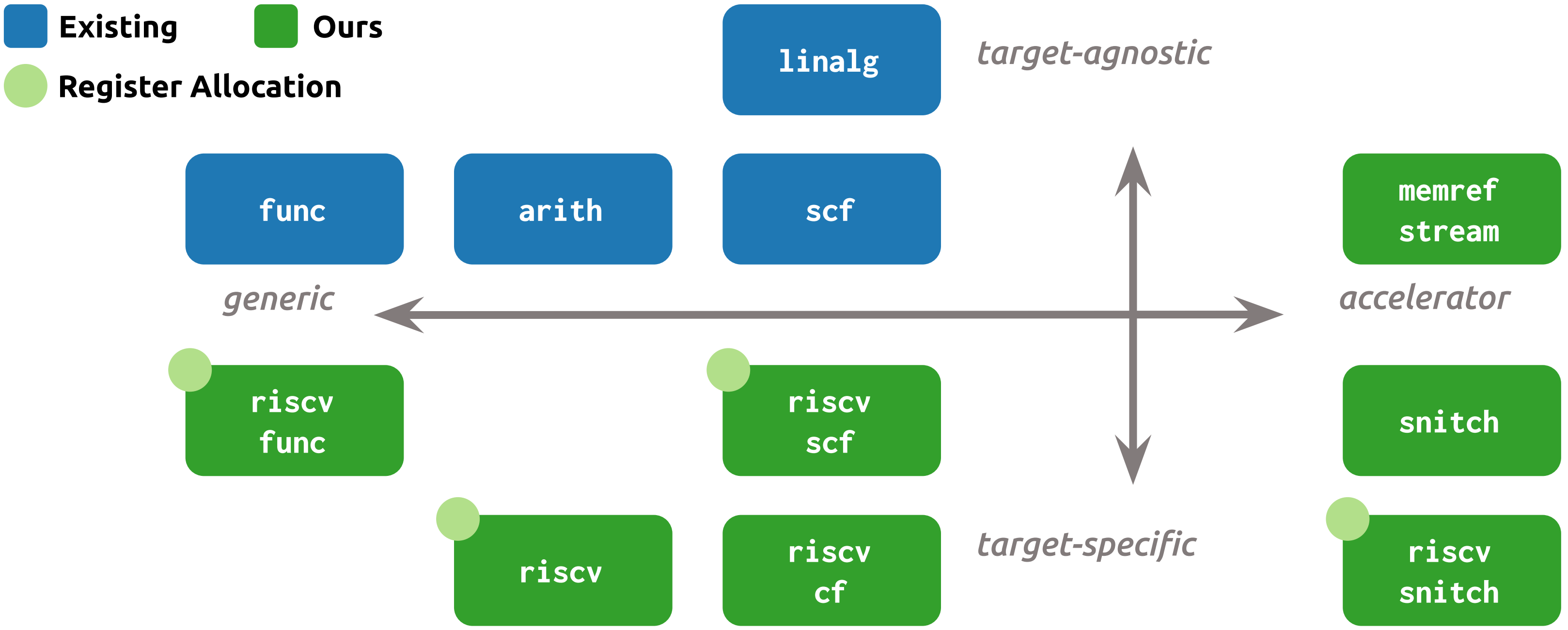
A Multi-Level Compiler Backend

xdsl.dev



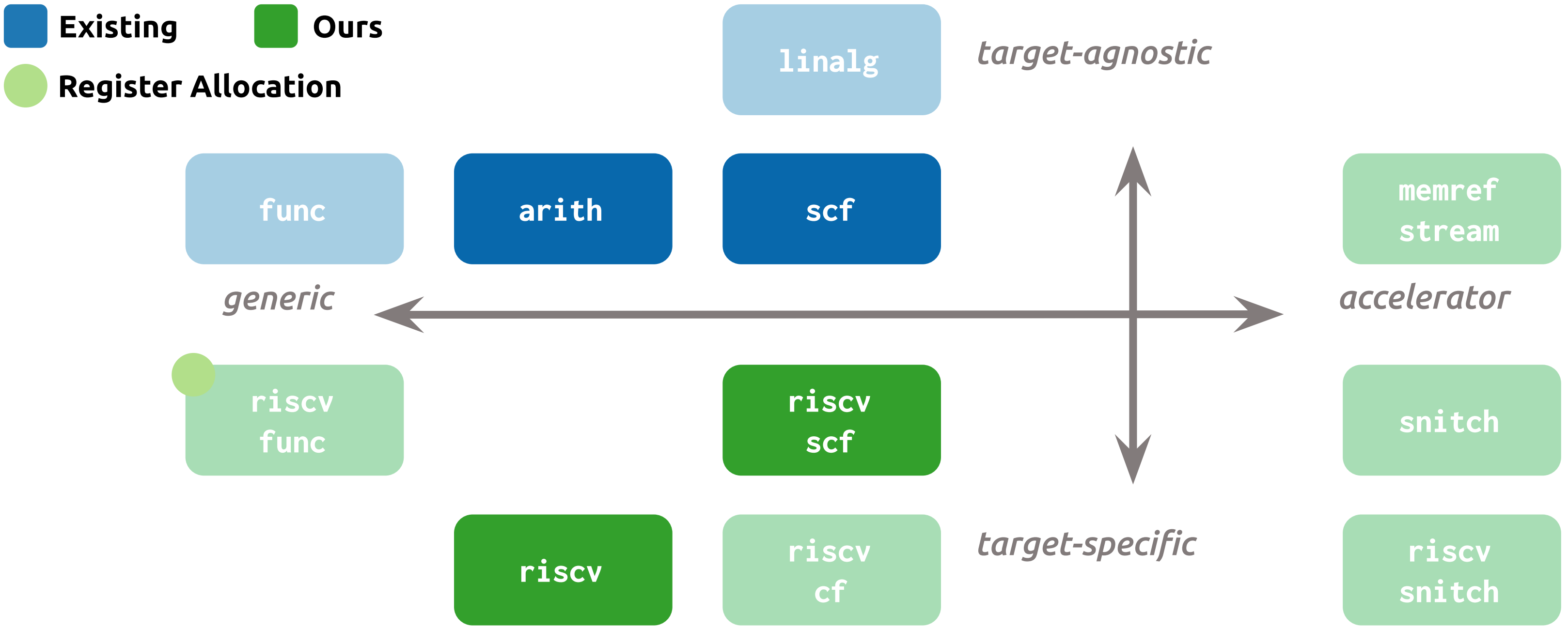
A Multi-Level Compiler Backend

xdsl.dev



A Multi-Level Compiler Backend

xdsl.dev



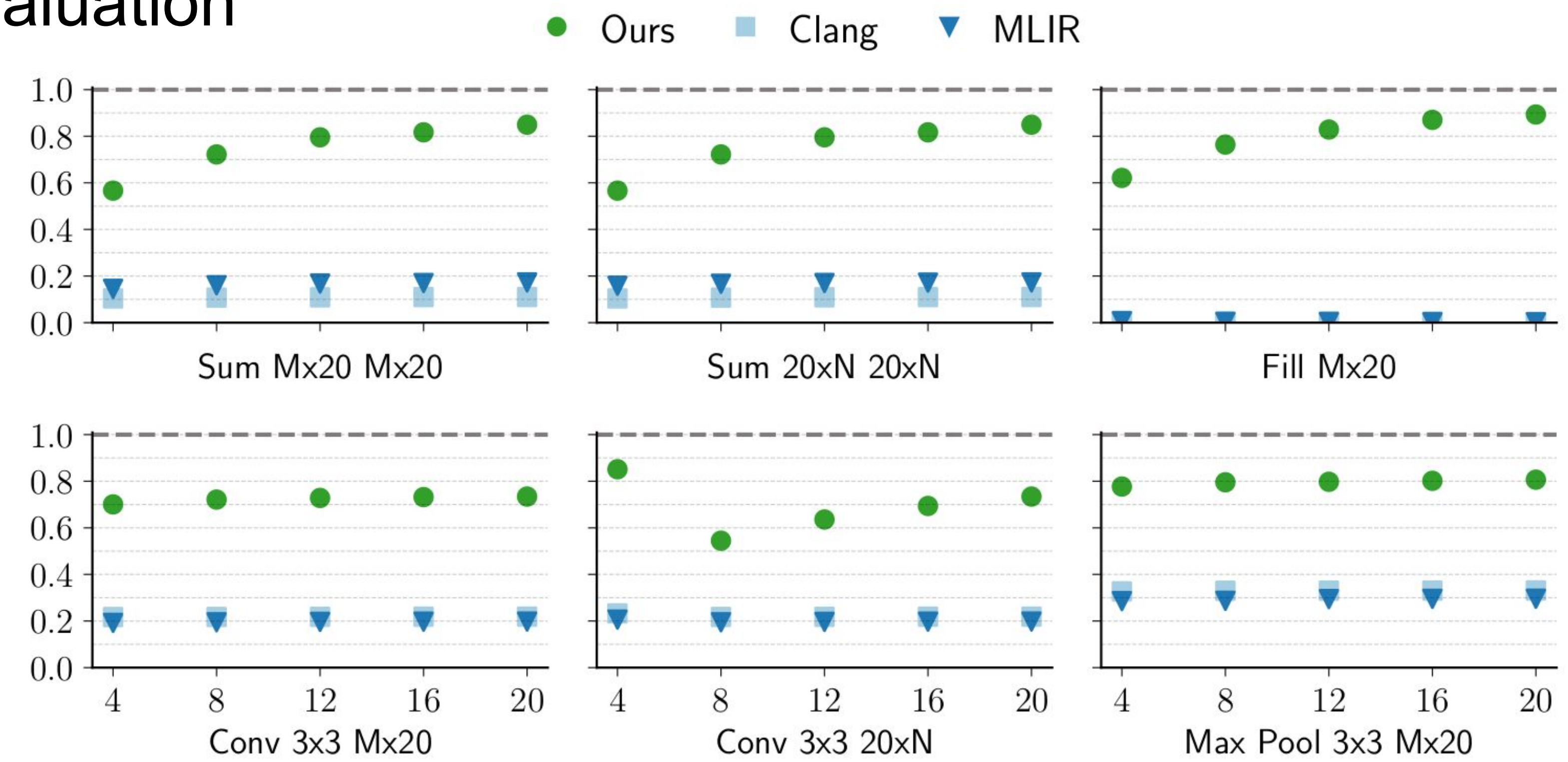
scf+arith dialects

```
%c0   = arith.constant 0 : index
%c128 = arith.constant 128 : index
%c1   = arith.constant 1 : index

scf.for %arg4 = %c0 to %c128 step %c1 {
  %x = memref.load %arg1[%arg4] : memref<128xf64>

  %y = memref.load %arg2[%arg4] : memref<128xf64>


  %z = arith.addf %x, %y : f64
  memref.store %z, %arg3[%arg4] : memref<128xf64>
}
```

riscv dialects

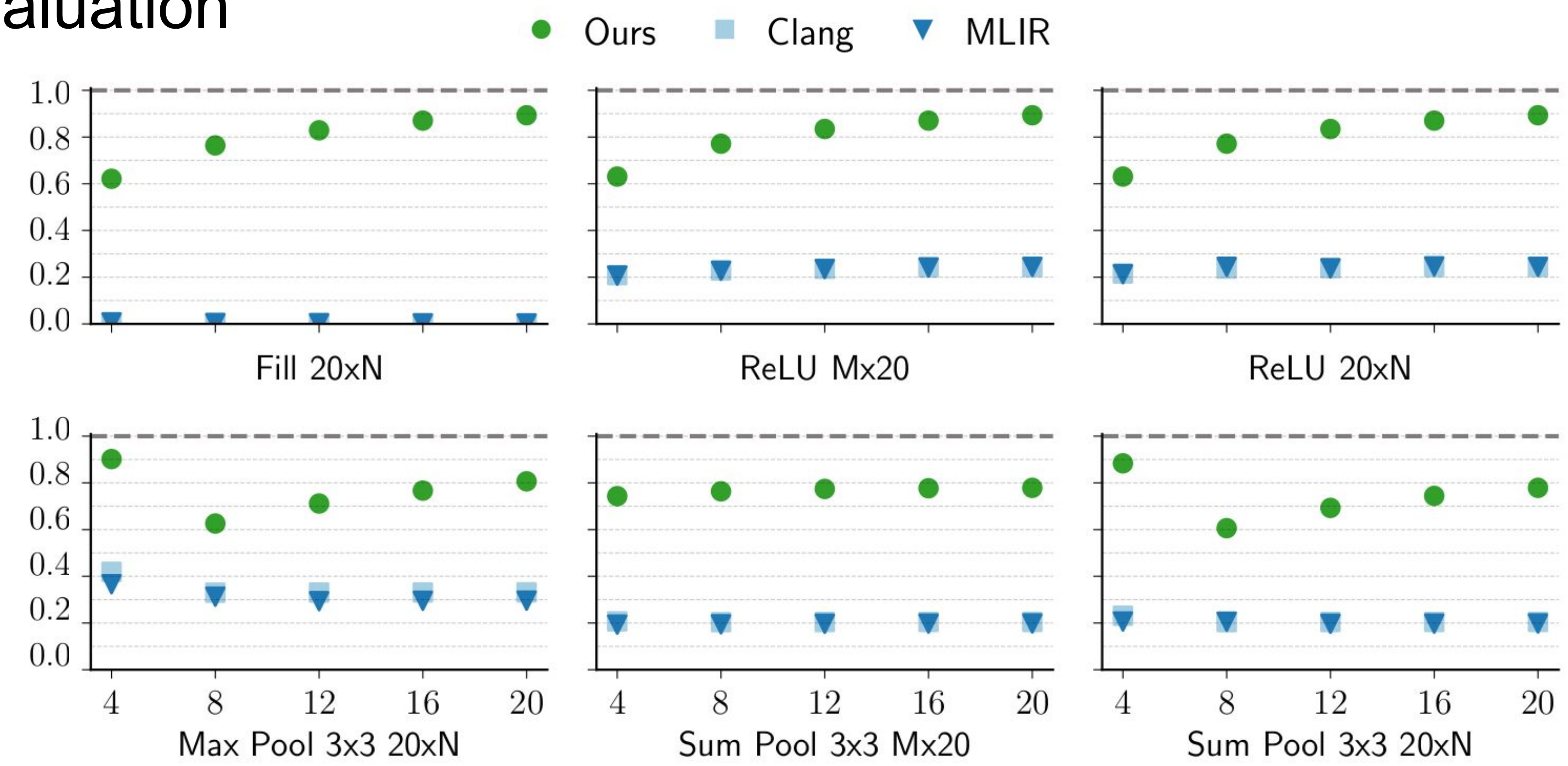
```
%c0    = riscv.li 0 : !riscv.reg<>
%c1024 = riscv.li 1024 : !riscv.reg<>
%c8    = riscv.li 8 : !riscv.reg<>

riscv_scf.for %arg4 : !riscv.reg<> = %c0 to %c1024 step %c8 {
  %xi = riscv.add %arg0, %arg4 : !riscv.reg<>
  %x  = riscv.flw %xi, 0 : !riscv.freg<>
  %yi = riscv.add %arg1, %arg4 : !riscv.reg<>
  %y  = riscv.flw %yi, 0 : !riscv.freg<>
  %zi = riscv.add %arg2, %arg4 : !riscv.reg<>
  %z  = riscv.fadd.d %x, %y : !riscv.freg<>
  riscv.fsw %zi, %z, 0
}
```

Evaluation



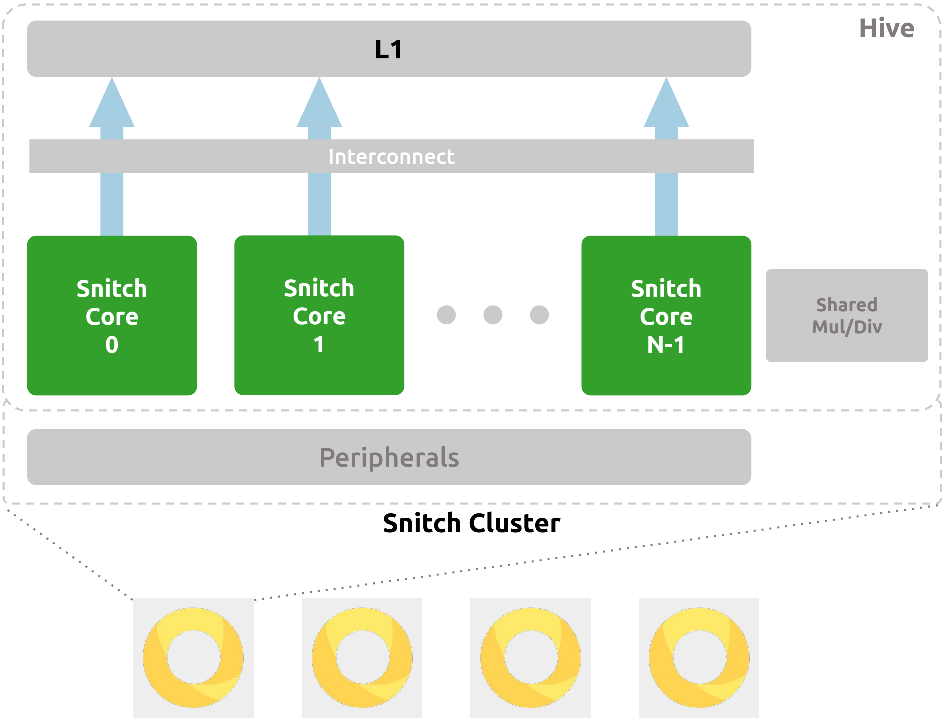
FPU utilization

Evaluation

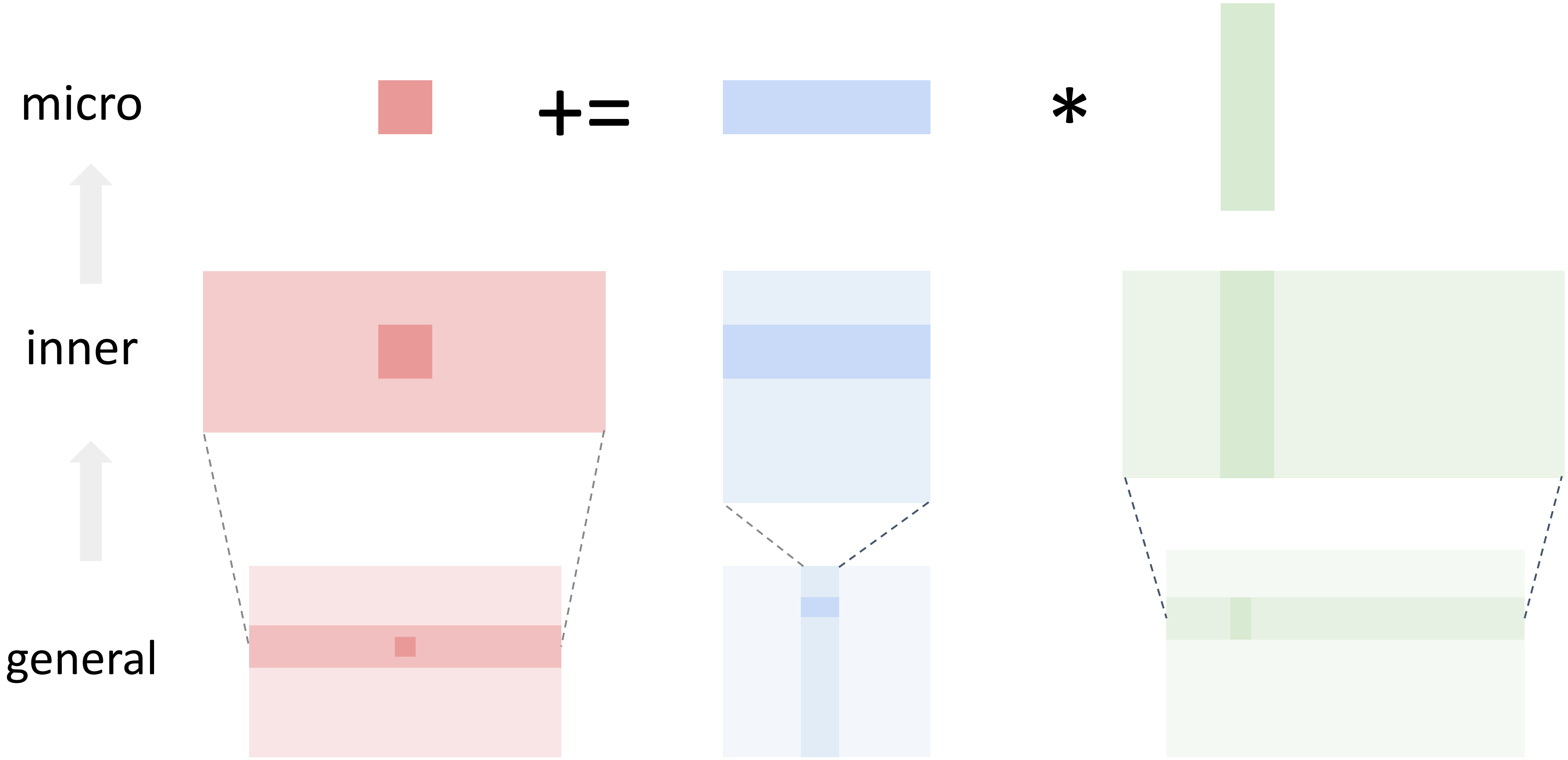


FPU utilization

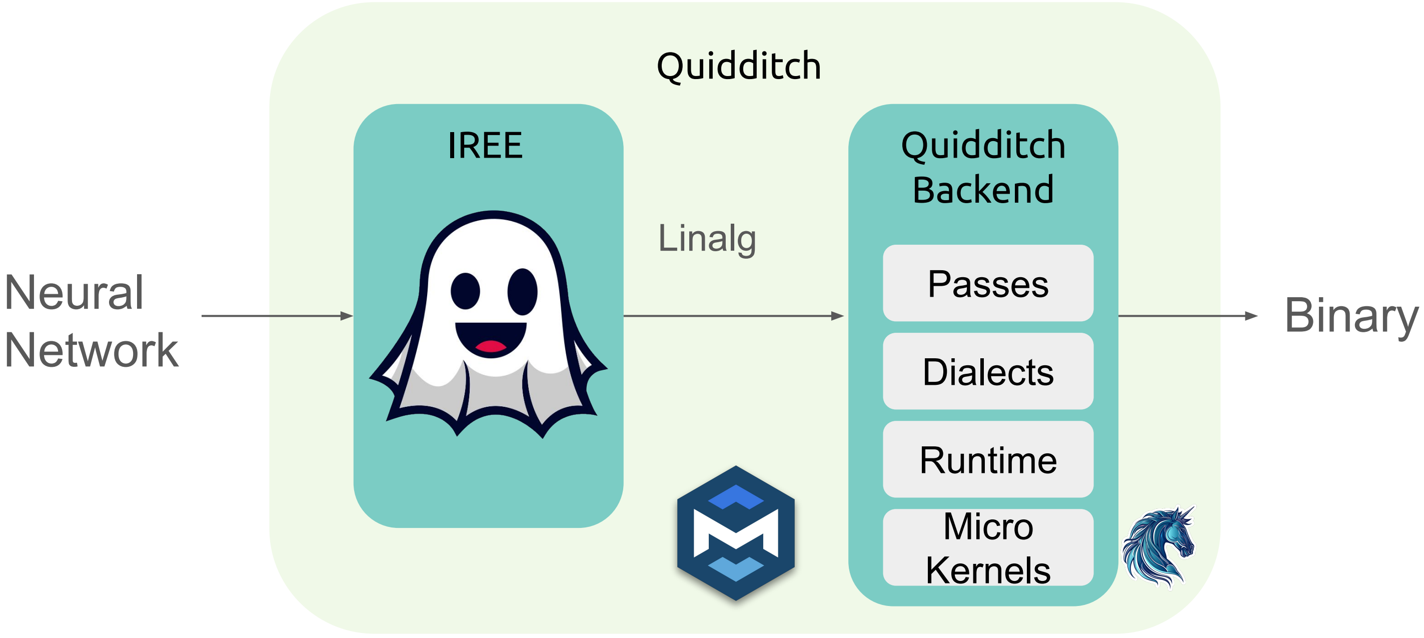
How do we distribute across clusters?



View over the Memory Hierarchy

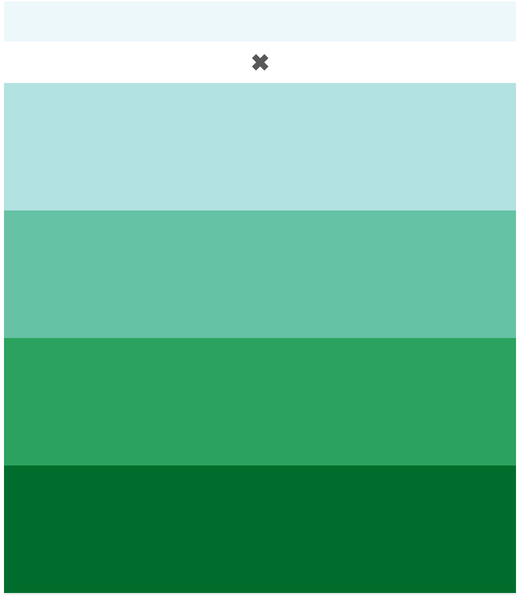


Quidditch Compiler

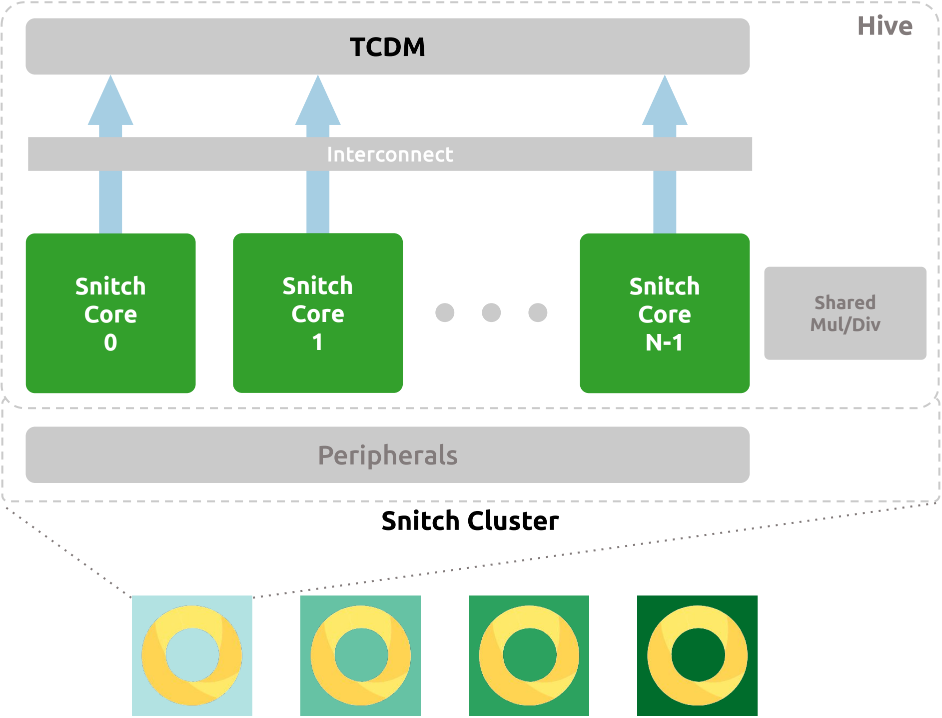


github.com/opencompl/quidditch

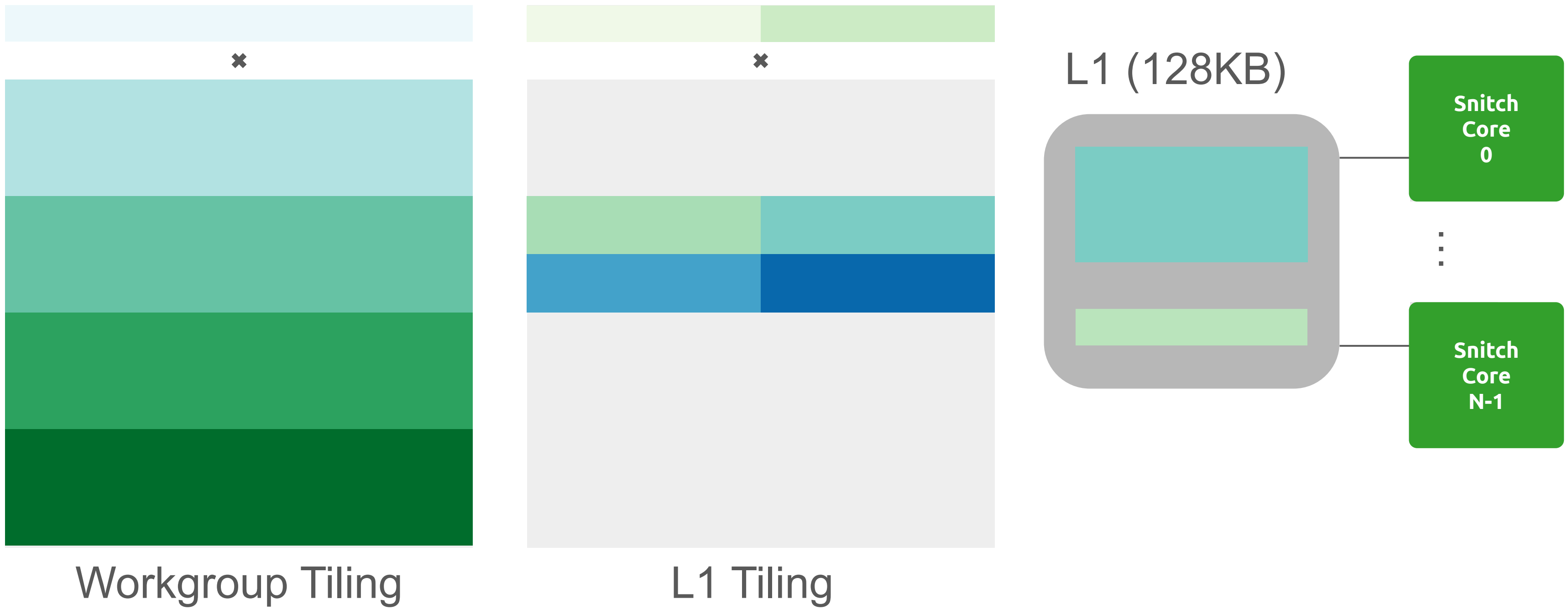
Workgroup Tiling



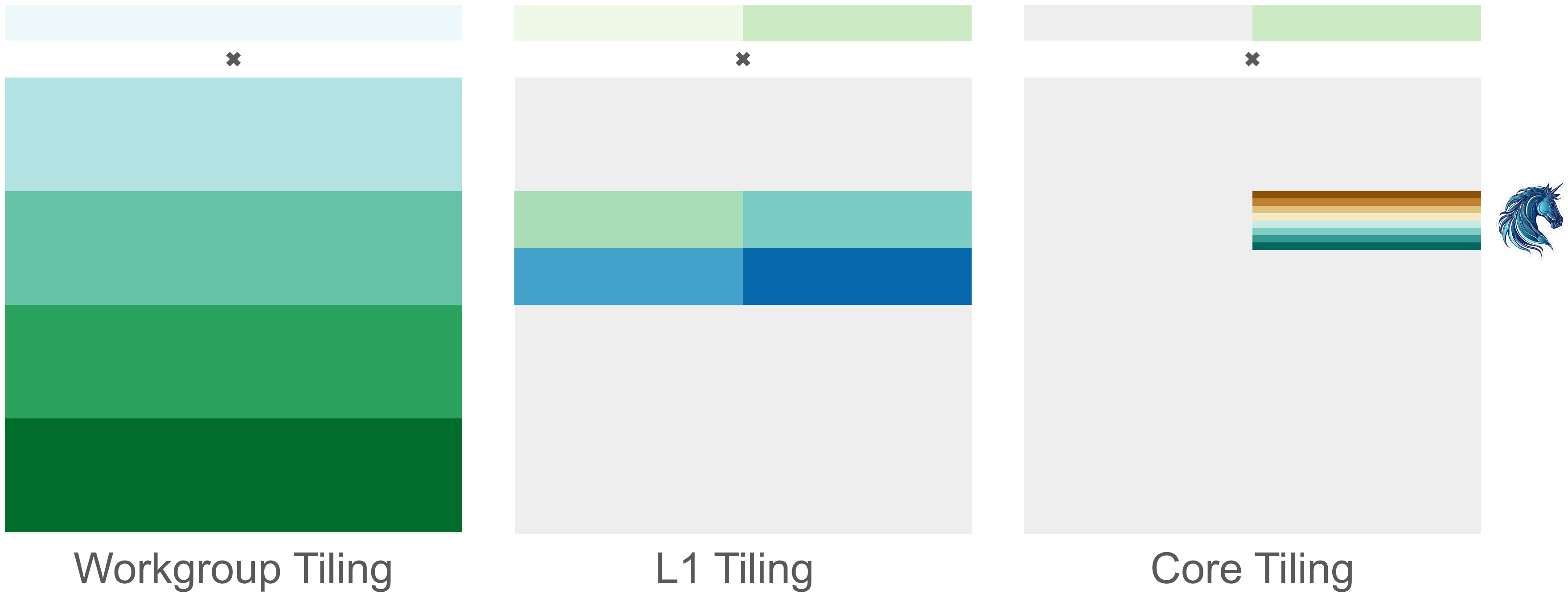
Workgroup Tiling



Tiling across the Memory Hierarchy

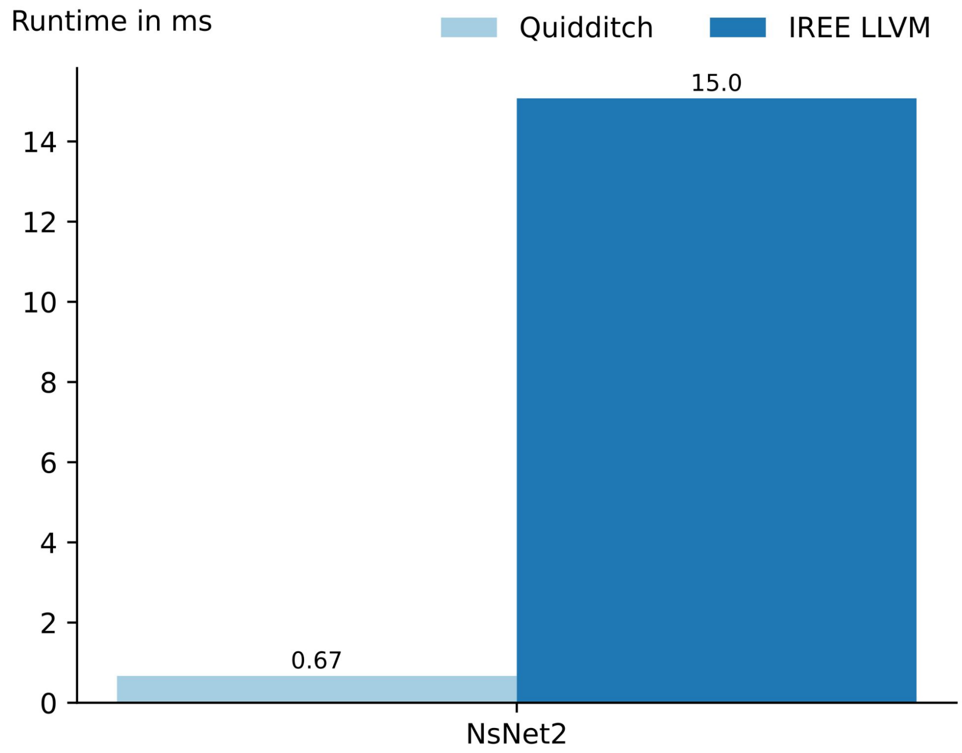


Tiling across the Memory Hierarchy



NsNet2 on a Snitch Cluster (8 cores)

22x speedup



NN Compilers for Custom HW Accelerators

- Progressively tailored abstractions can maximize concurrency and utilization of custom accelerators

- Efficient memory management leveraging hierarchy characteristics and constraints xDSL

ExCALIBUR

README License

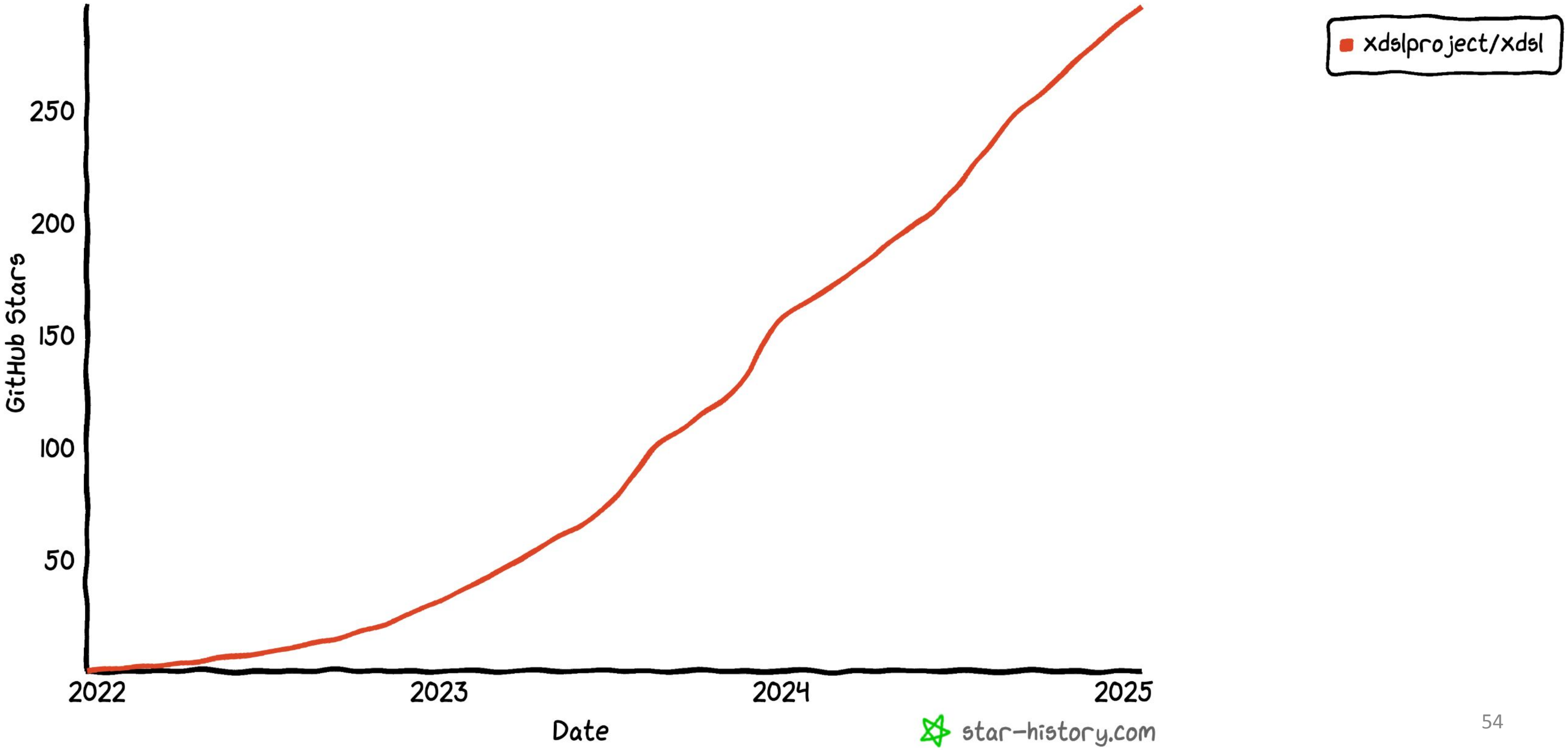
CI - Python-based Testing passing | pypi package 0.23.0 | downloads 387k
downloads/week 21k | codecov 90% | chat on zulip

xDSL: A Python-native SSA Compiler Framework

xDSL is a Python-native compiler framework built around SSA-based
intermediate representations (IRs). Users of xDSL build a compiler by

xDSL

ExCALIBUR



xDSL

xdsl.dev